Architecture

"Mathochist Studios" Cohort 4, Team 11

(Assessment 2)

Euan Cottam
Charlie Thoo-Tinsley
Harri Thorman
Will King
Zach Moussallati
Aiden Turner
Marcus Williamson
Joshua Zacek

Design Languages and Tools

To create representations of our product's architecture, we used the Unified Modeling Language (UML), using the PlantUML language to create UML diagrams using text instead of manually drawing them. We used PlantText, an online PlantUML text editor, to create and render these diagrams.

Entities

Initially, our entities were classed as either *dynamic* or *static*. This approach is key to our system as some entities will move and some will be fixed in place, requiring different business logic i.e. moving logic vs. collision logic. This is ostensibly a limited architecture, yet is sufficient for our game due to its simplicity i.e we don't need complex entities or interactions. Figure 1 shows an abstracted view of our dynamic-static architecture.

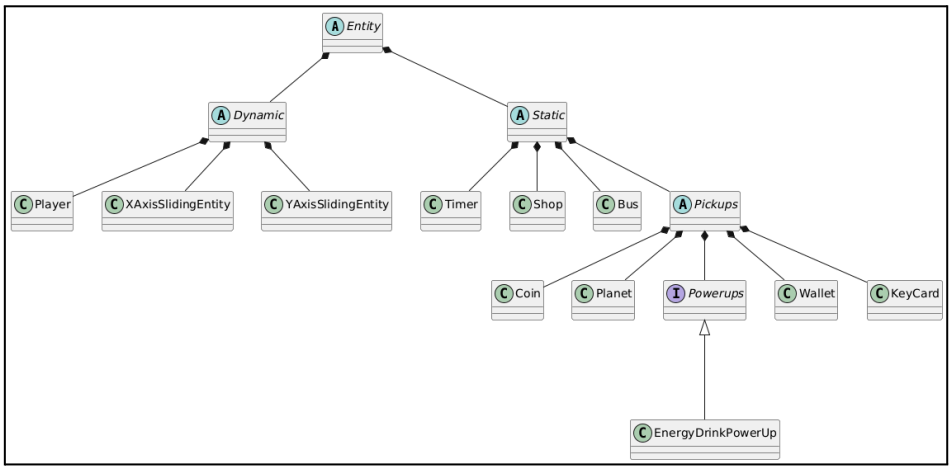

Figure 1: Abstracted UML class diagram for entities

Figure 2 shows the entities that remain stationary throughout the entirety of the game, meaning most of their values come from their position and collisions with other objects. These static entities consist of collectables ("Pickups") and features of different events in the game mainly used to aid the player in completing the game in some kind of way.

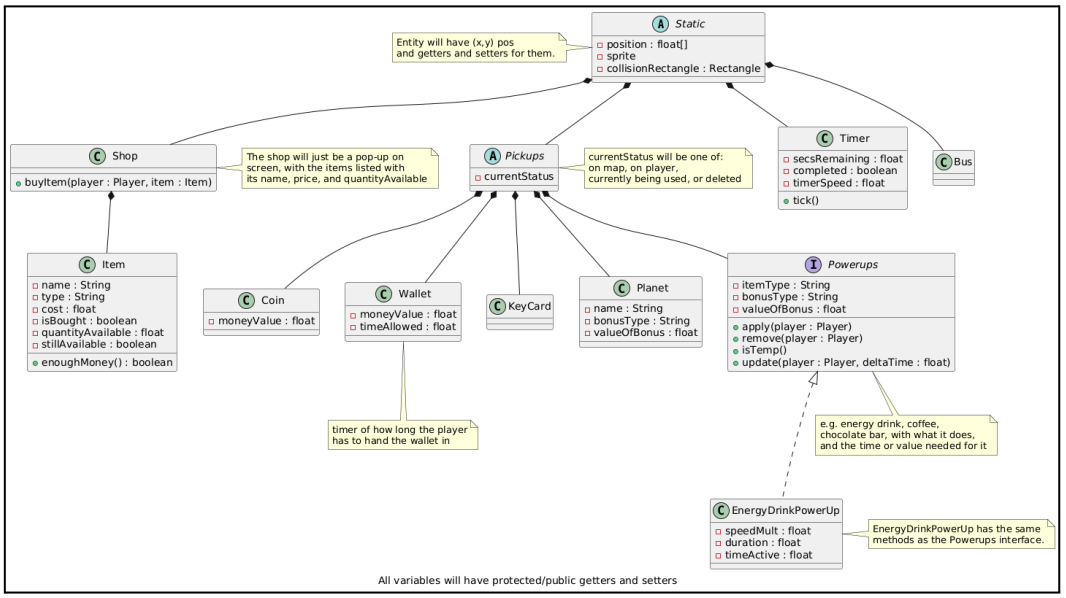

Figure 2: Detailed UML class diagram for static entities

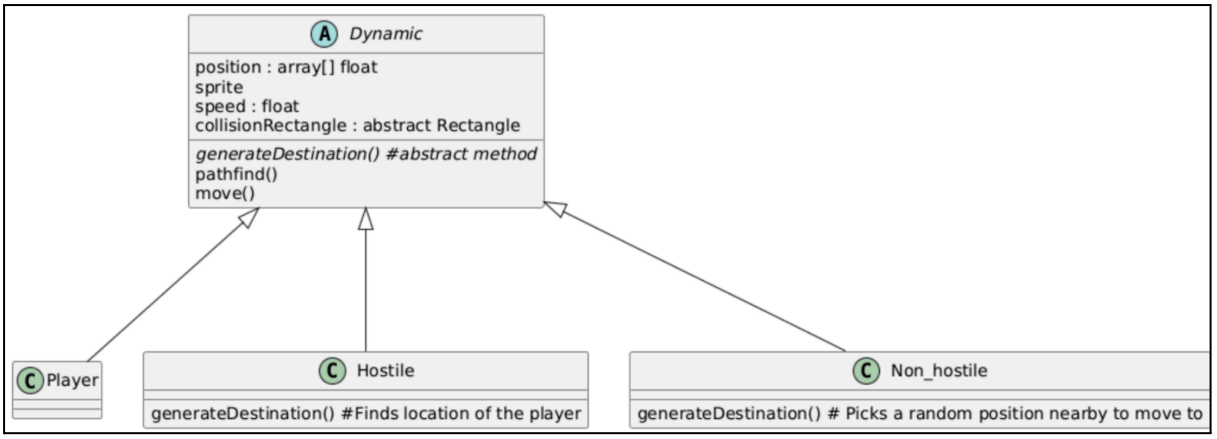

Figure 3: Abstract UML class diagram of the first iteration dynamic entity architecture

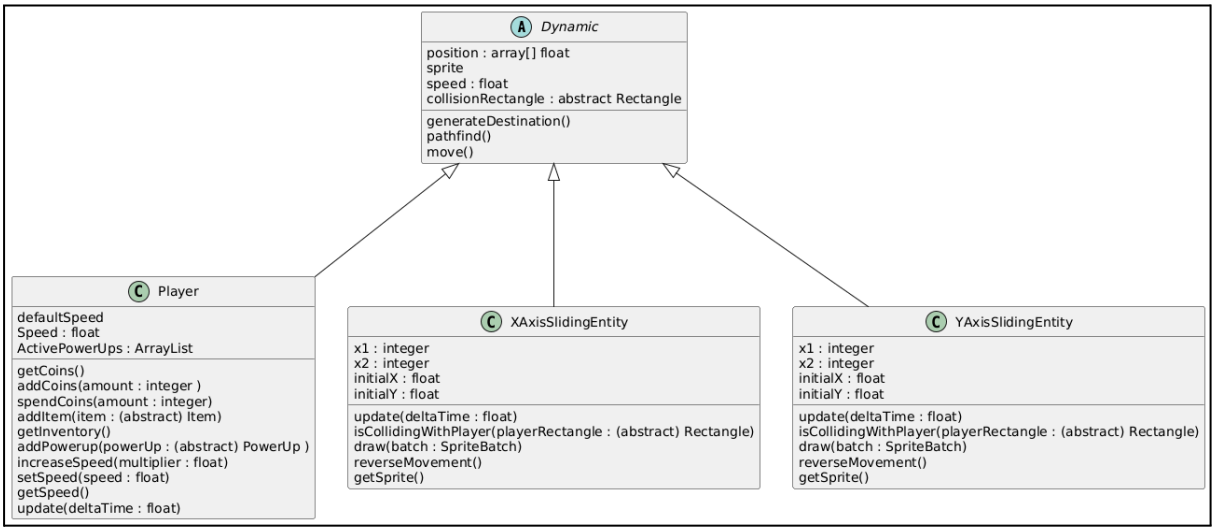

Figure 4: Detailed UML class diagram of a later iteration of dynamic entity architecture

Figures 3 and 4 represent the architecture for the dynamic entities. These entities can move around the world, meaning they have speeds, pathfinding and positions which can constantly change. Though our current design doesn't feature any moving objects, it is still useful to have this framework should a dynamic object (e.g. enemies, NPCs) need to be implemented later down the line.

Figure 3 shows an initial design for the dynamic objects. This design thinks of the dynamic objects as either hostile (chases the player) or non-hostile (friendly).

The only difference between non-hostile and hostile entities is in the *generateDestination()* method, hence why it is an abstract method in the *Dynamic* abstract class. For hostile objects, *generateDestination()* returns the *(x,y)* position of the player, and will constantly move to this position - imitating a chase. Conversely, non-hostile objects will just move randomly around a space as they won't chase you. This design, however, was scrapped as the team favoured a simpler dynamic object implementation. The current implementation sees moving entities categorised into ones that either move solely in the *X axis*, solely in *Y axis* or are *stationary*. Figure 4 shows the design of this interpretation of Dynamic entities.

At later stages, we decided this architecture model for our entity was unfit for purpose; We didn't find stationary and moving entities to be a meaningful distinction. We decided to use

an Object-Oriented architecture with several separate types of entities: InteractableEntity, Enemy, and XAxisSlidingEntity (legacy).

We kept XAxisSlidingEntity because there were some entities from earlier stages of development that we wanted to continue using in their current form, and refactoring them to use Enemy would've been too time consuming. In our final architecture, only *one* entity uses this legacy Entity type.

The remaining two Entity types have two important distinctions: Enemy is triggered by a player collision with a specified radius defined by EnemyAI, while InteractableEntity is triggered by a player key press within a specified radius of the InteractableEntity. Additionally, Enemy uses EnemyAI to handle movement, whereas InteractableEntity is static by default i.e. it must define its own movement. EnemyAI system makes this easy to implement and modular i.e. you can use one EnemyAI with multiple Enemy entities. This is particularly useful for EnemyAIs with complicated movement algorithms, allowing for decomposition.

You can see these differences reflected in Figures 5 and 6, where InteractableEntity has its radius defined in the main interface (interactionRadius: float), while Enemy has its radius defined as a private attribute in an EnemyAI. Additionally, you can see the "static-as-default" behavior of InteractableEnemy in entities such as *BirdSeed*, which have movement-related methods and attributes. Although there are InteractableEntity entities which can move i.e. Bus, but they need to define their own movement. In contrast, Enemy passes this responsibility to EnemyAI.

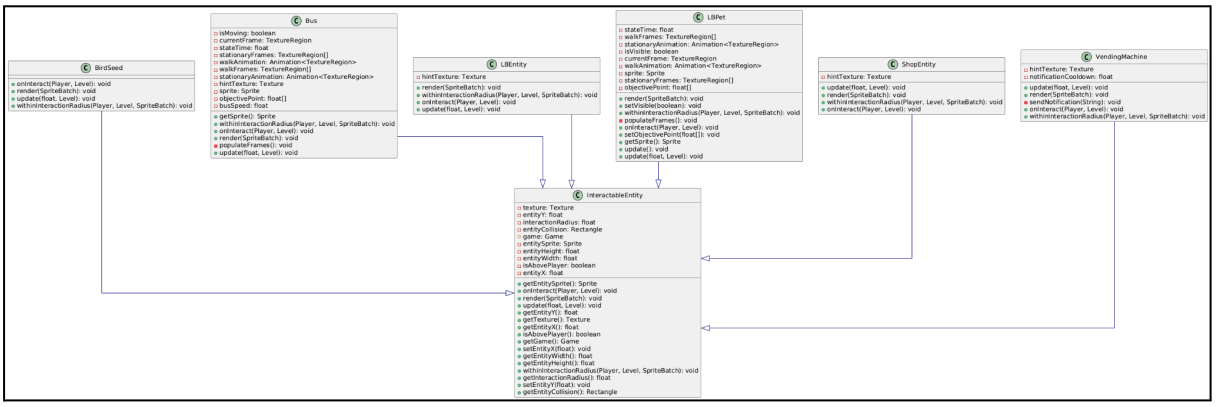

Figure 5: Detailed UML class diagram of InteractableEntity

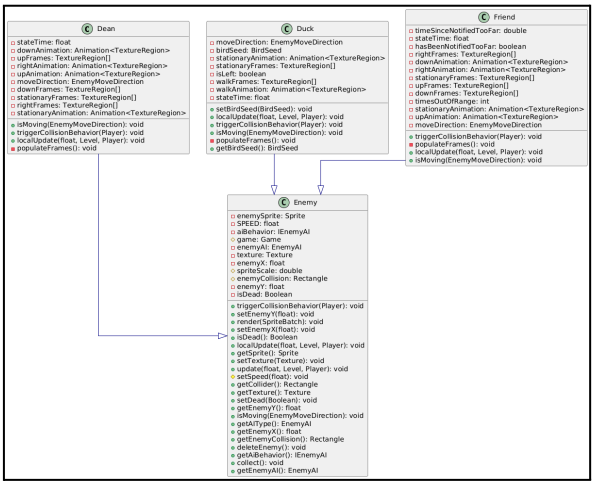

Figure 6: Detailed UML class diagram of Enemy

Figure 7: Detailed UML class diagram of EnemyAI

Maps and Viewport

As per *UR_UNIVERSITY_ACCURATE*, we need to create a university-themed map/world. We're using Tiled software to create TMX files. Handily LibGDX provides a loader to read files generated by Tiled.

Initially, our map was separated into 4 distinct regions which represent different areas of the university. The benefit of this was that it was easier to divide workload amongst the team i.e. each member does their level and all of the events/entities in their respective region. Furthermore, the game is drawing and rendering less to the screen at one time, reducing the computational workload on the user's device. The viewport is fixed at the center of each region such that the player can view the region in its entirety. A static viewport reduces the complexity of the system.

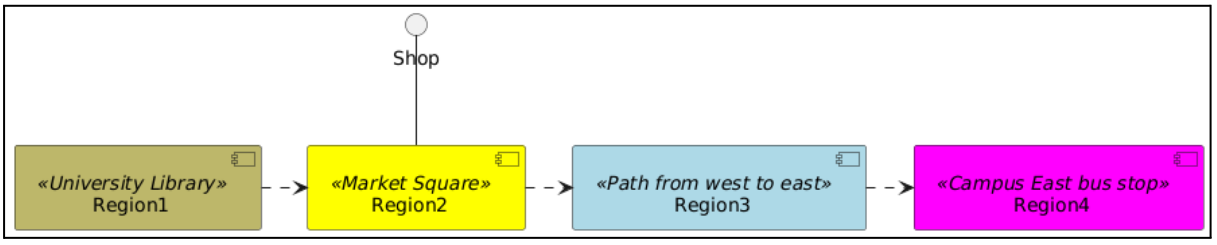

Figure 8: UML component diagram of the map regions

Figure 8 portrays how each region followed on from each other initially and the different timepoints of gameplay. Every region had a unique interface based on a university landmark (NFR_UNIVERSITY_ACCURATE_LOOK). The shop is a separate level which has purchasable items dispersed around the level. Simply colliding with the items will buy them.

At later stages, we decided that this model, while on the right track, was too restrictive, especially after conducting user testing, which indicated to us that the "forward-exclusive" directionality frustrated users.

As a result, while keeping the "region" model, we decided to make the regions bi-directional, so that you could go back and forth. Combined with additional content in each region, modification of the win condition (see next section), and a more "maze-like" experience, user exploration was incentivized, enhancing the gameplay experience. Figure 9 represents this change in our "region" model.

<INSERT Figure 9>

Win Condition

Initially, the player could complete the game by taking the bus. To take the bus, the player needed to have a sufficient balance, achieved through collecting coins throughout the map. At later stages, we decided to still use the bus as a win condition, but forfeit the requirement for coin collection. We decided to offset this by adding further "obstacles" earlier in the map. Figure 10 & 8 show these differing philosophies.

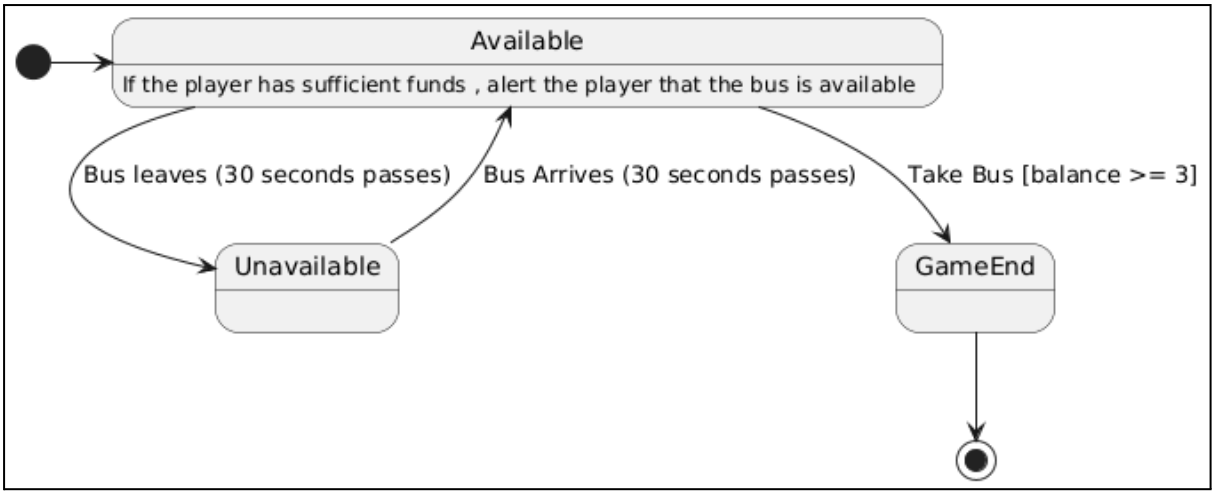

Figure 10: UML state transition behavioural diagram showing initial win condition

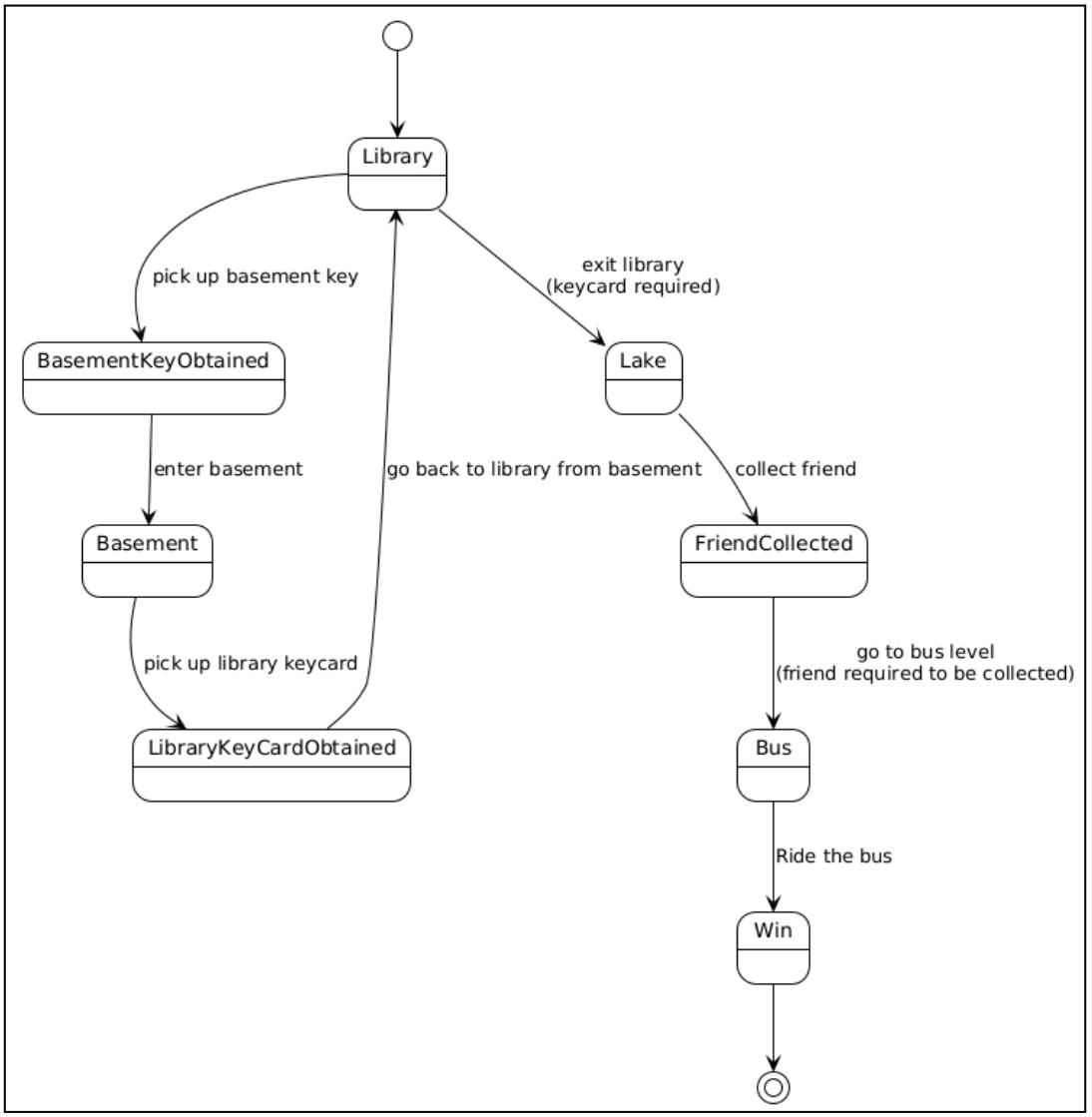

Figure 11: UML state transition behavioural diagram showing revised win condition and added "obstacles" to account for such

Movement and Collision

Our system needs to feature simple movement mechanics (UR_CASUAL_FRIENDLY solved by FR_EASY_MOVEMENT_CONTROLS, NFR_CLEAR_BOUNDARIES) as well as collision. We're using the LibGDX input module to detect input. Figure 12 shows how collisions and movement are processed.

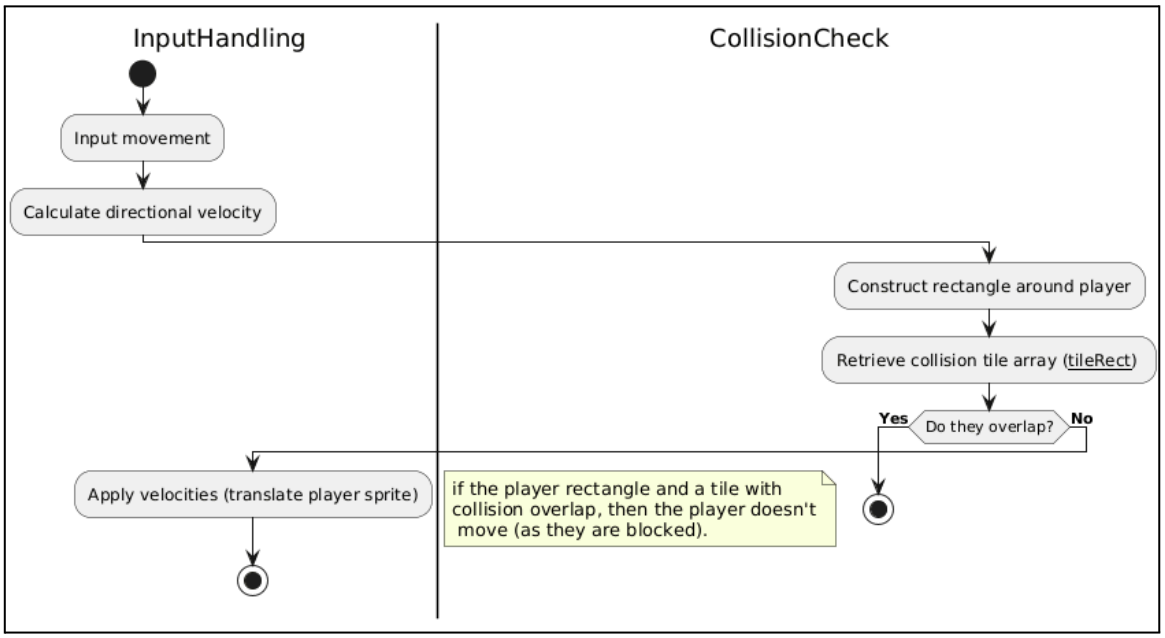

Figure 12: UML activity diagram for movement and collisions

CRC cards

PLAYER	
Collecting a coin	Wallet
Colliding with another entity	collisionRectangle
Able to move in all directions	position, speed

SHOP	
Items can only be purchased with the required amount of money	Wallet
Items available to buy with gives a certain powerup	ShopItem, Powerups
Only sell items in stock	ShopItem, quantityAvailable()

ENEMY	
Will seek to collide with player	pathFind(), Player
Able to move in all directions	position, speed

PICKUPS	
Pickups can be collected (they disappear) once a player collides with it.	collisionRectangle, currentStatus
Specific pickups give the player a bonus	Planets, PowerUps, valueOfBonus, bonusType
Wallet value increases every time a coin is picked up	Coin, Wallet, Player